MODULAR REASONING FOR ACTOR SPECIFICATION DIAGRAMS

Scott F. Smith[1] and Carolyn L. Talcott[2]

[1]The Johns Hopkins University*
scott@cs.jhu.edu

[2]Stanford University†
clt@cs.stanford.edu

Abstract: Specification diagrams are a novel form of graphical notation for specifying open distributed object systems. The design goal is to define notation for specifying message-passing behavior that is expressive, intuitively understandable, and that has formal semantic underpinnings. The notation generalizes informal notations such as UML's Sequence Diagrams and broadens their applicability to later in the design cycle. In this paper we show how it is possible to reason rigorously and modularly about specification diagrams. An Actor Theory Toolkit is used to great advantage for this purpose.

INTRODUCTION

Specification diagrams are a novel form of graphical notation for specifying open distributed object systems. Our goal is to define notation for specifying message-passing behavior that is expressive, intuitively understandable, and that has a rigorous underlying semantics. Many specification languages that have achieved widespread usage have a graphical presentation format, primarily because engineers can understand and communicate more effectively by graphical means. Popular graphical specification languages include Universal Modeling Language (UML) and its predecessors [16], Petri nets, and Statecharts [9]. UML is the now-standard set of object-oriented design notations; it includes several different forms of graphical specification notation. Our

*Partial funding provided by NSF grants CCR-9312433 and CCR-9619843
†Partial funding provided by ONR grant N00014-94-1-0857, NSF grant CRR-9633419, DARPA/Rome Labs grant AF F30602-96-1-0300, and DARPA/SRI subcontract 17-000042.

aim is a language with similar intuitive advantages but significantly greater expressivity and formal underpinnings. The language is also designed to be useful throughout the development process, from an initial sketch of the overall architecture to detailed specifications of final components that may serve as formal documentation of critical aspects of their behavior. The underlying communication assumptions we use are taken from the actor model: object- and not channel-based naming is used, open systems are treated explicitly, and message passing is asynchronous, fair, and with nondeterministic arrival order.

Rather than attempting an overview of the language here, the best introduction is via examples, which we try to get quickly to. The next section is a brief introduction to the underlying actor communication model, and this is followed by a short introduction to the diagram syntax. The following section shows how specification diagrams may be used in practice via examples. Then, the actor theory framework and toolkit is outlined. This is a general framework for defining and reasoning about actor systems. Following is a section outlining how diagrams may be given operational meaning via an actor theory, and lastly we conclude with an example proof of correspondence between diagrams. By necessity many technical details are missing from this abbreviated presentation; see [18] for a more complete exposition. [17] is the initial paper on this topic.

Actor Concepts

We provide a brief overview of the underlying actor basis at this point. The Actor Theory Section below builds the actor theory framework over this basis. Actors are distributed, object-based message passing entities. Since actors are object-based, they each have a unique *name*, and actors may dynamically create other actors. Individual actors independently compute in parallel, and actors only communicate by message passing. Messages are sent asynchronously, so message send instructions never block. All messages must eventually arrive at their destination, but with arbitrary delay.

To describe interactions with an actor system component we assume given a set of *actor names* $a \in \mathbf{A}$ and a set of *messages* $M \in \mathbf{Msg}$. These constitute an *actor communication basis*. Message packets *mp* are of the form $a \triangleleft M$, indicating message M is sent to actor a. Actor names may occur in messages and entities of other kinds used in modelling actor computation. To determine which names occur we require that each such set X be equipped with an acquaintances function $acq : X \to \mathbf{P}_\omega(\mathbf{A})$ giving the finite set of actor names occurring in elements of X. We also assume that the set of acquaintances can be renamed without changing the structure or meaning of such an entity.

Actor systems are intended to model open distributed computation. This means that the whole system will not be explicitly specified, and the framework must assume some *external actors* are interacting with the local system. Additionally, of the local actors, only some of their names may be known by external entities; these are the *receptionists*. We use ρ to denote the set of receptionists and χ to for the set of external actors of an actor system component. The pair (ρ, χ) is called the system *interface* and we write S^ρ_χ to indicate that S is an open actor system with receptionists ρ and known external actors χ.

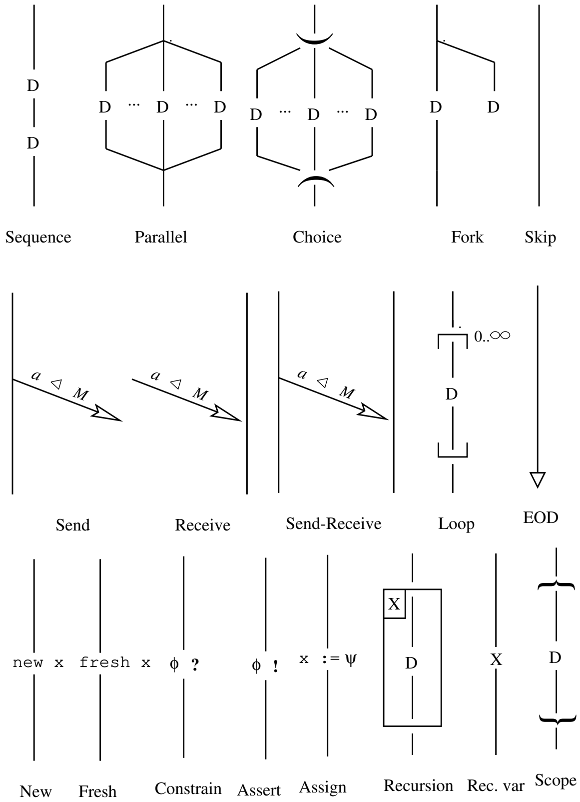

Figure 1 Specification Diagram Components

An individual "run" of an actor system is modeled by an *interaction path*, that is, a possibly infinite sequence of interactions: inputs, $\mathtt{in}(a \triangleleft M)$, and outputs, $\mathtt{out}(a \triangleleft M)$. The interface of an interaction path is that of the system. In such a run the set of receptionists will grow if new local names are sent out in messages and the set of external actors will grow when previously unknown names are received in messages from the outside. The *interaction path law* requires that a message can only be input to a receptionist or output to a known external actor.

SYNTAX

In this section we present the syntax of specification diagrams, and an informal description of their meaning. We use two forms of notation for diagrams, one graphical and one textual. The graphical one is intended for use in practice: the graphical drawings are highly intuitive. However for mathematical study the textual form is easier to manipulate. Figure 1 presents the graphical diagram components. Vertical lines indicate progress in time going down, expressing abstract causal ordering on events, with events above necessarily leading to events below. This causal ordering will be termed a *causal thread*. Note there is no necessary connection between these "threads" and actors or processes, the threads exist only at the semantic level: a single thread of causality may be multiple actors, and a single actor may have multiple threads of causality. In the figure, D itself may be any diagram, and any two components may be connected to any other via a vertical line to form a sequenced diagram. Figure 2 in the next section presents some examples.

Before describing the constructs one-by-one, we define the form that the atomic units may take: diagram state variables x, constraints ϕ, assignments ψ, message expressions mp_{d}, and actor expressions a_{d}. The set $\mathbf{X}_{\mathrm{d}}$ is the set of diagram state variables x, y, z, $\ldots$ used in diagrams. These variables may take on values in a fixed mathematical universe $\mathbf{U}$ which we do not completely specify. Diagram messages $\mathbf{Msg}_{\mathrm{d}}$ are messages $\mathbf{Msg}$ that may have state variables ocurring in them, and similarly $\mathbf{A}_{\mathrm{d}}$ is either a state variable x or actor name a. The assignment expression ψ is a function on $\mathbf{U}$ which may refer to variables in $\mathbf{X}_{\mathrm{d}}$. Predicates on $\mathbf{U}$ are notated ϕ. $acq(D)$, the actor names known to D, is defined as $\bigcup_{u \in \mathbf{U}} \{acq(u) \bigm| u$ occurs as an $M_{\mathrm{d}}, a_{\mathrm{d}}, \phi$ or ψ in $D\}$.

The individual graphical constructs are now informally described. With each construct the textual grammatical equivalent is given in parentheses.

sequence ($D_1;D_2$) Vertical lines (causal threads) represent necessary temporal sequencing of events in D_1 before those in D_2.

parallel ($D_1 \mid D_2$) Events in parallel diagrams have no causal ordering between them, but are after events above and before events below.

choice ($D_1 \oplus D_2$) One of the possible choices is taken. There is no requirement that the choice be fair, in the sense that for a particular actor computation the same branch could always be taken. Standard if-then-else is easily definable with choice and constraints.

fork ($\mathtt{fork}(D)$) A diagram is forked off which hereafter will have no direct causal connection to the future of the current thread (however, messages could indirectly impose some causality between the two).

skip (`skip`) Does nothing.

send ($\mathtt{send}(a \triangleleft M)$) A message is sent to a with contents M. There is a requirement that message delivery be fair, in the sense that any message sent must eventually arrive at its destination.

receive ($\mathtt{receive}(a \triangleleft M)$) A message is received by actor a, possibly binding pattern variables in the message M, which can be used below in the diagram. This statement blocks until a message arrives matching its pattern. If a message arrives but never matches any `receive`, that message starves, and the computation path is thus considered unfair and is not admitted.

send-receive (discussed below) A message is sent from one component of the diagram to another, producing a causal cross-connection in what could have been causally unrelated segments. The edge imposes a strong constraint, because it means the sender must have sent the message to the indicated receiver, and it was explicitly received.

loop ($\mathtt{[}D\mathtt{]}^{0\ldots\infty}$) The diagram is iterated some number n times, where n is nondeterministically chosen from the interval $0\ldots\infty$. The case $n=\infty$ chosen means it loops forever. The textual syntax here is not in the core language—it is a macro abbreviat-

ing $\texttt{rec}X.((D;X)\oplus\texttt{skip})$. Loop $[D]^{0\ldots\omega}$ is similar but cannot run forever, and $[D]^{\infty}$ must only run forever.
EOD (`eod`) Denotes the end of a causal thread in the diagram; expressed by existing syntax $\texttt{rec}X.(\texttt{skip};X)$.
scope ($\{x_0,\ldots,x_n : D\}$) Brackets demarcate static scoping of state variables. In the official textual syntax explicit variable declarations are given (and the variables initially given arbitrary values), but by an an implicit convention discussed below, bracketing alone may be used to define variable extent.
recursion ($\texttt{rec}X.D$**,** X) A boxed diagram fragment may refer to itself by name, X, so X occurring inside the box refers to the whole box. $X \in \mathbf{X}_\mathrm{r}$ is a countable set of recursion variables. Standard while-do syntax may be defined via recursion, choice, and constraints.
new ($\texttt{new}(x)$) State variable x is assigned arbitrary contents.
fresh ($\texttt{fresh}(x)$) State variable x is given contents consisting of an actor name not currently in use.
constraint ($\texttt{constrain}(\phi)$) An arbitrary constraint is placed on the current thread of causality, which must be met. There is no direct analogue to this notion of constraint in programming languages: the constraint may be any mathematical expression, and a constraint failing does not indicate a run-time error, it indicates that such a computation path will not arise. This interpretation is analogous to the **assume** predicate of Dijkstra's predicate calculus [6].
assertion ($\texttt{assert}(\phi)$) An arbitrary assertion is made. Unlike `constrain`, an `assert` that fails indicates failure of some property, and has no programmatic meaning. This is analogous to Dijkstra's **assert** predicate.
assign($x := \psi$) A variable is dynamically assigned a new value. The assignment body, ψ, can be any sensible mathematical expression.

We define some syntactic sugar which allows variable declarations at a scope boundary to be implicit: $\{D\}$ abbreviates $\{x_0,\ldots,x_n : D\}$ for $x_0,\ldots,x_n$ being all state variables occurring directly in D (not in a deeper lexical level) Certain syntax is easily encodable via macros and so is not defined in the core grammar: `abort` abbreviates $\texttt{constrain}(\mathit{false})$; `fail` abbreviates $\texttt{assert}(\mathit{false})$; and, $\texttt{new}(x = u)$ abbreviates $\texttt{new}(x);\texttt{constrain}(x = s)$. Translation from graphical diagrams into textual notation is obtained by inductively replacing the graphical syntax with the corresponding textual syntax listed above. A global transformation is used to encode the constraint introduced by cross-edges, a topic we must skip here.

A *top-level* specification diagram includes an interface, notated $\langle D\rangle^{\rho}_{\chi}$. Top-level diagrams are modules which may be directly given semantic meaning. We will not always include the phrase "top level" but meaning should be clear from context.

USING SPECIFICATION DIAGRAMS

In this section we aim to illustrate the full range of functionality of the language via examples. The subsequent sections outline how the meaning of diagrams and asserted properties of diagrams may be established.

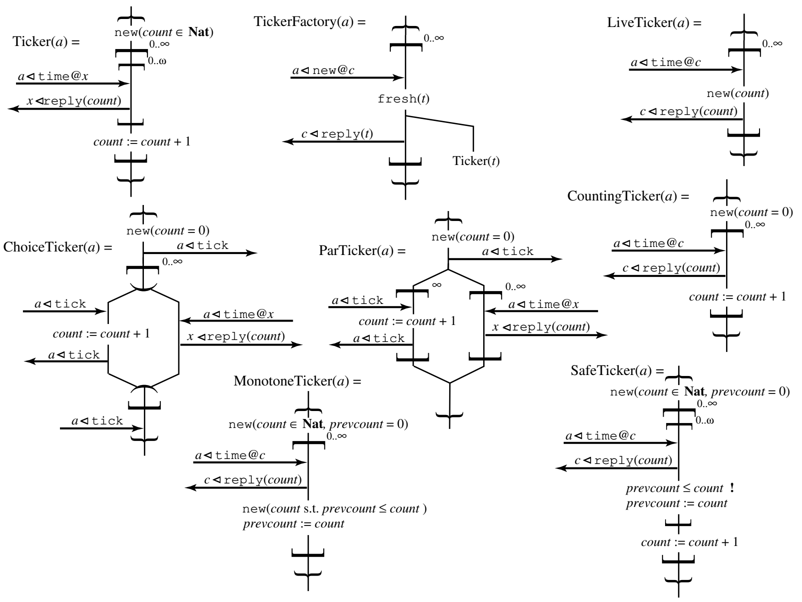

Figure 2 Ticker Specifications

Expressing patterns of message passing

We give here a series of examples illustrating how the constructs of the language may be used to build interesting specifications.

Ticker. A ticker is a simple specification of a monotonically increasing counter which replies to `time` messages with its current count value. This example illustrates the unbounded nondeterminism present in actor computations. A high-level specification is as follows. This high-level specification as well as the other ticker examples to follow appear diagrammatically in Figure 2.

$$\begin{array}{l}\mathrm{Ticker}(a) = \\ \{\ \mathtt{new}(\mathit{count} \in \mathbf{Nat}); \\ \mathtt{[}\ \mathtt{[}\ \mathtt{receive}(a \triangleleft \mathtt{time@}x);\ \mathtt{send}(x \triangleleft \mathtt{reply}(\mathit{count}))\mathtt{]}^{0\ldots\omega};\ \mathit{count}\ \mathtt{:=}\ \mathit{count} + 1\ \mathtt{]}^{0\ldots\infty}\ \}\end{array}$$

This succinct specification expresses the fact that the count can stay constant for finitely many `time` requests, but then must increment. The meaning of this top-level diagram, $[\![\langle \mathrm{Ticker}(a)\rangle^{a}_{\emptyset}]\!]$, is a set Ip_{Ticker} of interaction paths *ip*. The diagram meaning function $[\![\cdot]\!]$ is outlined in the Operational Semantics Section below. Figure 2 also contains the TickerFactory, a factory for producing tickers, which shows how new actors may be dynamically created via `fresh` and `fork`.

Function Composer. We define a distributed method for computing $g \circ f$ for composable functions f and g: the functions are computed at remote nodes and the values combined at a third node. For any actor name, a, and function $f \in V \to W$, the following diagram $\mathrm{F}(a, f)$ specifies a component that accepts requests to a of the form

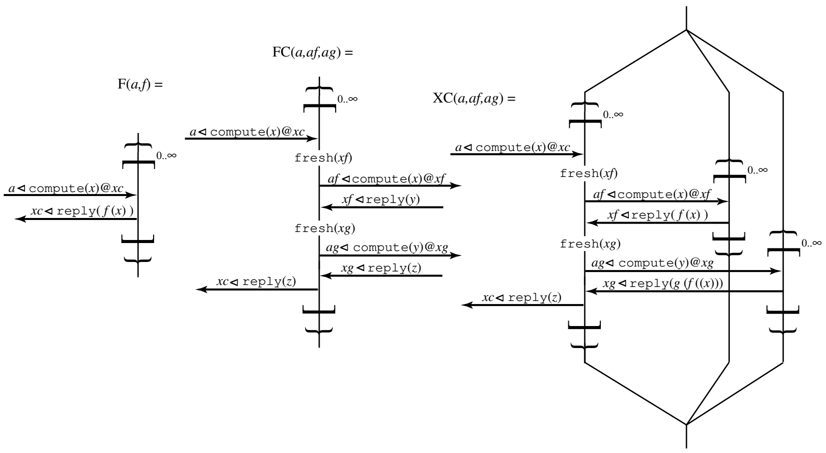

Figure 3 Function Composer Specifications

$\texttt{compute}(v)@c$ and sends to c a $\texttt{reply}(f(v))$.

$$\mathrm{F}(a,f) = \{x,xc : \texttt{[receive}(a \triangleleft \texttt{compute}(x)@xc);\ \texttt{send}(xc \triangleleft \texttt{reply}(f(x)))\texttt{]}^{0\ldots\infty}\}$$

Graphical diagrams of this and the other function composer examples are given in Figure 3. For any actor names a, af, ag, $\mathrm{FC}(a, af, ag)$ specifies a component that accepts requests to a of the form $\texttt{compute}(v)@c$, asks af to `compute` on v, and then asks ag to `compute` on the result from af, sending that result to c.

$$\begin{array}{l}\mathrm{FC}(a,af,ag) = \{\ \texttt{[}\\ \texttt{receive}(a \triangleleft \texttt{compute}(x)@xc);\texttt{fresh}(xf);\ \texttt{send}(af \triangleleft \texttt{compute}(x)@xf);\\ \texttt{receive}(xf \triangleleft \texttt{reply}(y));\texttt{fresh}(xg);\ \texttt{send}(ag \triangleleft \texttt{compute}(y)@xg);\\ \texttt{receive}(xg \triangleleft \texttt{reply}(z));\ \texttt{send}(xc \triangleleft \texttt{reply}(z))\texttt{]}^{0\ldots\infty}\ \}\end{array}$$

The fresh names xf and xg are private names FC sends as the customer to the target only; the target is to reply to this name, guaranteeing the reply came from the target or the target's accomplice. Thus in a context where af is the name of an f computer and ag is the name of a g computer, FC composes af and ag to become a $g \circ f$ computer. $\mathrm{C}(a, f, g, af, ag)$ puts FC in such a context:

$$\mathrm{C}(a,f,g,af,ag) = (\mathrm{FC}(a,af,ag) \mid \mathrm{F}(af,f) \mid \mathrm{F}(ag,g))$$

$\mathrm{XC}(a, f, g, af, ag)$ of Figure 3 is nearly identical to $\mathrm{C}(a, f, g, af, ag)$, with the exception that the send and receive edges are connected cross-edges. This may not seem like much of a difference, but the cross-edge diagram makes a much stronger assertion about the message-passing behavior in that it constrains how sends and receives must line up at run-time. This is the main reason for the presence of send-receive cross edges in the diagram syntax. XC is thus a higher-level specification, and C is an equivalent one closer to implementation.

Relating Specification Diagrams

In this section, terminology for when an implementation satisfies a specification will be informally introduced. For this we need a useful notion of $\langle D_I\rangle^{\rho}_{\chi} \models \langle D_S\rangle^{\rho}_{\chi}$ (implementation D_I satisfies specification D_S). Two concrete notions of satisfaction are now presented.

Loose Satisfaction. The standard notion of satisfaction in the literature is that the implementation refines the specification. We term this *loose satisfaction*. In algebraic specification for example [20], this is the only relation defined to assert an implementation meets a specification.

This form of satisfaction is a subset relationship on interaction path behaviors:

Definition 1 (loose satisfaction): $\langle D_I\rangle^{\rho}_{\chi} \models \langle D_S\rangle^{\rho}_{\chi}$ iff $[\![\langle D_I\rangle^{\rho}_{\chi}]\!] \subseteq [\![\langle D_S\rangle^{\rho}_{\chi}]\!]$.

Loose Satisfaction and the Ticker. Loose specifications may easily be written in specification diagrams. Consider for instance the LiveTicker(a) of Figure 2. This specification of the ticker only specifies that all time requests receive a reply. The loose satisfaction relation $\langle \mathrm{Ticker}(a)\rangle^{a}_{\emptyset} \models \langle \mathrm{LiveTicker}(a)\rangle^{a}_{\emptyset}$ then is in effect asserting the liveness of the Ticker. Diagrams are in fact extremely useful as a means whereby properties of specifications may be asserted, a topic that will be addressed more fully shortly.

Another important use of loose satisfaction is in cases when it is necessary to have an implementation more deterministic than the specification. An example is the CountingTicker(a) of Figure 2. This implementation replies with a number one bigger, loosely satisfying the Ticker specification: $\langle \mathrm{CountingTicker}(a)\rangle^{a}_{\emptyset} \models \langle \mathrm{Ticker}(a)\rangle^{a}_{\emptyset}$.

Strong Satisfaction. For an implementation to fully and faithfully satisfy a specification, the two must have the same sets of interaction paths, i.e., their observable behavior to the environment is identical. This is *strong* satisfaction, an equivalance.

Definition 2 (strong satisfaction): $\langle D_I\rangle^{\rho}_{\chi} \models\!\mid \langle D_S\rangle^{\rho}_{\chi}$ iff $[\![\langle D_I\rangle^{\rho}_{\chi}]\!] = [\![\langle D_S\rangle^{\rho}_{\chi}]\!]$.

Strong Satisfaction of the Function Composer. A good example of strong satisfaction is found in the context of the function composer example above. A high-level specification for computing $g \circ f$ is just $\mathrm{F}(a, g \circ f)$ which directly computes $g \circ f$. We may then assert the following.

Theorem 3. $\langle \mathrm{C}(a, f, g, af, ag)\rangle^{a}_{\emptyset} \models\!\mid \langle \mathrm{F}(a, g \circ f)\rangle^{a}_{\emptyset}$

This Theorem will be proved in the penultimate Section. The proof shows how it is possible to rigorously reason about diagrams.

Strong Satisfaction, Restriction, and the Ticker. A more low-level specification of the ticker could send `tick` messages to itself to increment the counter: every time it receives a `tick`, it increments the counter and sends itself another `tick`. One possible manner of writing this is the ChoiceTicker of Figure 2. Another reasonable alternative is to use parallelism between the `tick` and `time` instead of nondeterministic choice, the ParTicker of Figure 2. The Ticker, ChoiceTicker, and

ParTicker are close to being equivalent specifications. However, the Ticker does not accept `tick` messages, so it is in fact slightly different. Nonetheless, in a context where actor a receives only `time` messages, all three are equivalent. For this purpose we use a restriction operator to restrict interactions to relevant paths. Letting V be $\{a \triangleleft \mathtt{time}@c \mid c \in \mathbf{A} \text{ and } c \neq a\}$, the notation $\langle \text{ChoiceTicker}(a) \rangle^{a}_{\emptyset} \lceil V$ indicates the top-level diagram in a context where input messages are from V only.

Theorem 4. $\langle \text{ChoiceTicker}(a) \rangle^{a}_{\emptyset} \lceil V \;\; \models\!\!\mid \;\; \langle \text{ParTicker}(a) \rangle^{a}_{\emptyset} \lceil V \;\; \models\!\!\mid \;\; \langle \text{Ticker}(a) \rangle^{a}_{\emptyset}$.

Note that $\langle \text{Ticker}(a) \rangle^{a}_{\emptyset} \lceil V \;\; \models\!\!\mid \;\; \langle \text{Ticker}(a) \rangle^{a}_{\emptyset}$, since Ticker($a$) does not accept `tick` messages,

Asserting Properties of Specifications Diagrammatically

Safety and liveness properties can be asserted directly in the specification diagram language. We present three different techniques for asserting safety and liveness properties diagrammatically. The first method is based on loose satisfaction. The second is based on diagrammatically defining an environment which enforces the specification to have the proper behavior. The third method is by directly decorating the specification with logical assertions.

The first method is as follows. The underlying idea is that a predicate (such as a liveness condition) can be defined by a diagram $P_{\mathrm{D}}(ip)$ iff $ip \in [\![\langle D \rangle^{\rho}_{\chi}]\!]$. The assertion that all behaviors of a particular diagram D' have the property P_{D} is then just $\langle D' \rangle^{\rho}_{\chi} \models \langle D \rangle^{\rho}_{\chi}$.

An example of this method was in fact given earlier: the property that that all `time` messages sent to the Ticker will receive a reply was expressed by the LiveTicker specification, and the statement $\langle \text{Ticker}(a) \rangle^{a}_{\emptyset} \models \langle \text{LiveTicker}(a) \rangle^{a}_{\emptyset}$ asserts the Ticker has such a property. It is trivial to show that LiveTicker responds to all time messages, so satisfaction of this specification is the liveness argument.

A second and perhaps more convincing way to assert liveness is by specifying an environment which enforces liveness. For the Ticker, the environment should assert that all `tick` requests are handled. The `assert` predicate is used for this purpose (more precisely the `fail` macro, defined as `assert`(*false*)) in the following LiveTickerEnvt:

$$\text{LiveTickerEnvt}(a) = [\mathtt{fresh}(c); \mathtt{send}(a \triangleleft \mathtt{time}@c); (\mathtt{receive}(c \triangleleft x) \oplus (\mathtt{fail};\ \mathtt{eod}))]^{0 \ldots \infty}$$

Failure arises only when there is a `time` request that does not get answered – the `receive` choice is never possible and so failure is the only possibility. If failure were chosen when there was in fact a reply to be received, that path would be unfair because a message was never received. We then assert $\models \langle \text{Ticker}(a) \mid \text{LiveTickerEnvt}(a) \rangle^{\emptyset}_{\emptyset}$, which means there are no paths of this combined system which contain a `fail` event.

Safety properties are properties that hold at certain points in execution. As with liveness, safety can both be asserted by showing the specification satisfies an abstract specification which obviously has safety. The third manner in which properties may be diagrammatically asserted is that safety properties may be directly asserted in the specification itself via `assert` decorations. An example of a specification decorated with a safety assertion is a ticker which asserts successive outputs are non-decreasing.

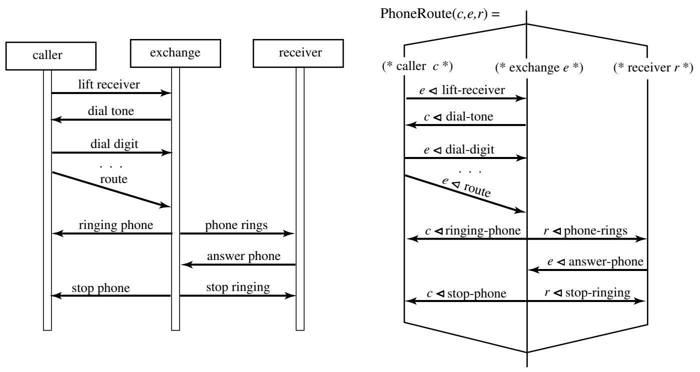

Figure 4 A UML Sequence Diagram and its Corresponding Specification Diagram

Then, the safety assertion is $\models \langle \text{SafeTicker}(a) \rangle^{a}_{\emptyset}$, for SafeTicker of Figure 2. The ability to express assertions diagrammatically means there is less need to learn a specialized logic in which assertions are written.

Comparison with UML Sequence Diagrams

Specification diagrams were partly inspired by UML sequence diagrams [16] (to be specific, by an early version known as a message trace diagram [15]), and the two share concepts. However, sequence diagrams are primarily designed to show possible scenarios of execution, and not to give all possible scenarios. They are primarily for informal design and not rigorous specification.

The left diagram of Figure 4 is a UML sequence diagram which is presented in [4] (we have removed temporal constraints since specification diagrams are not real-time). On the right side is the same example re-expressed as a specification diagram. This example also brings out the usefulness of the send-receive cross-edges in specification diagrams. An approximate analogy of features and properties is found in the following list of "Sequence diagrams::specification diagrams" analogies: parallel vertical lifelines::parallel construct; split and merged lifelines::choice; asynchronous and synchronous messaging::asynchronous basis; incomplete informal semantics::complete, rigorous semantics; (not present)::usefulness as a logic for asserting properties; (not present)::explicit open systems modeling; activation/focus of control::(none – fundamentally concurrent); real-time constraints::(none currently); practical language::research language.

AN ACTOR THEORY FRAMEWORK

In this section we briefly review the semantic framework used to model actor systems; this framework will then be used to give semantics specification diagrams, to define satisfaction relations both between pairs of specification diagrams and to reason about these relations. More detail about the framework can be found in [19, 13].

Actor theories

The actor theory framework is a general mathamatical framework for defining and reasoning about the semantics of actor systems. It can be used to define operational semantics for actor programming languages or the specification diagram language (a main use here). But also very specialized actor theories may be defined, describing the behavior of one particular fixed actor system, for instance a "Ticker" actor theory. The principle effort in defining an actor theory is the definition of a set of reaction rules that describe how an actor system evolves internally. The framework then specifies the interaction paths induced by that rule set.

Actor theory structure. An actor theory contains an actor communication basis that provides sets of *actor names*, and *messages* equipped with acquaintance and renaming functions as described in the Introduction. It provides, in addition, a set of *actor states* and a set of *labelled reaction rules*. An actor state $\sigma \in \mathbf{S}$ describes a state of a group of one or more actors $iacts(\sigma) : \mathbf{P}_\omega(\mathbf{A})$ called the *internal actors*.

Reaction rule labels are used in deriving a labelled transition system semantics. Labels $l \in \mathbf{L}$ are equipped with four functions determining the (existing and new) actors and message packets participating in a transition. The *focus* actors $fActs(l) : \mathbf{P}_\omega(\mathbf{A})$ and received packets $rMPs(l) : \mathbf{M}_\omega(\mathbf{MP})$ are called *triggers*, elements that must be present for a rule with label l to apply. The created actors $cActs(l) : \mathbf{P}_\omega(\mathbf{A})$ and sent packets $sMPs(l) : \mathbf{M}_\omega(\mathbf{MP})$ are the *effects* of applying a rule with label l. For any label, the targets of received packets must be members of the focus actor set.

Reaction rules RR determine how the group as a whole evolves and responds to incoming messages. Each reaction rule is of the form $l : \sigma_0 \xrightarrow[cA,\mu_s]{fA,\mu_r} \sigma_1$ where $fA = fActs(l)$, $cA = cActs(l)$, $\mu_r = rMPs(l)$, and $\mu_s = sMPs(l)$. We omit mention of $fActs(l)$ if it is the set of internal actors of σ_0 and also omit mention of empty sets of created actors, received or sent message packets The set of rules must satisfy the fundamental actor locality laws of [1, 3].

Operational semantics of an actor theory. The operational semantics of an actor theory is given by a labelled transition relation on actor system configurations. A configuration has an interface and an interior. A *configuration interior*, I, is a state together with a multiset of message packets, called the *undelivered* messages. We write $\langle\sigma,\mu\rangle$ for the interior with state σ, and undelivered messages μ and I^{ρ}_{χ} for the configuration with interior I and interface (ρ,χ). The receptionists of a configuration must be a subset of the internal actors (of the state), the interface externals must be disjoint from the internal actors, and the acquaintances of the interior that are not internal actors must be a subset of the externals.

Configurations evolve either by internal computation according to the reaction rules, or by interaction with the environment. The transition rules are defined as follows.

Definition 5 (Transition rules):

$$\text{(internal)} \quad \langle\sigma_0,\mu\cdot\mu_r\rangle^{\rho}_{\chi} \xrightarrow{l} \langle\sigma_1,\mu\cdot\mu_s\rangle^{\rho}_{\chi} \quad \text{if} \quad l : \sigma_0 \xrightarrow[cA,\mu_s]{fA,\mu_r} \sigma_1 \in RR \wedge cA \cap (acq(\mu)\cup\chi) = \emptyset$$

(in) $\langle\sigma,\mu\rangle^{\rho}_{\chi} \xrightarrow{\mathtt{in}(a\triangleleft M)} \langle\sigma,\mu\cdot a\triangleleft M\rangle^{\rho}_{\chi\cup(acq(M)-\rho)}$ if $a \in \rho \wedge acq(M) \cap iacts(\sigma) \subseteq \rho$

(out) $\langle\sigma,\mu\cdot a\triangleleft M\rangle^{\rho}_{\chi} \xrightarrow{\mathtt{out}(a\triangleleft M)} \langle\sigma,\mu\rangle^{\rho\cup(acq(M)-\chi)}_{\chi}$ if $a \in \chi$

The set $\mathcal{P}$ of computation paths is the set infinite sequences of transitions of the form $\pi = [K_i \xrightarrow{tl_i} K_{i+1} \mid i \in \mathbf{Nat}]$. A computation segment is a finite segment of a computation path.

Constrained Actor Theories and Interaction Semantics. A *constrained actor* theory, *cATh*, is an actor theory *ATh* together with a subset $\mathcal{A}$ of the computation paths that are considered admissible in the derived interaction semantics. We write $\langle ATh, \mathcal{A}\rangle$ for the constrained actor theory with underlying actor theory *ATh* and admissible paths $\mathcal{A} \subseteq \mathcal{P}$. Admissibility can express notions of fairness, or require paths to be in a certain canonical form, or even express constraints on what the environment is allowed to do.

Interaction semantics gives a more abstract view of an actor system, specifying only the possible patterns of interaction that a system can have with its environment. The interaction semantics $[\![K : cATh]\!]$ of a configuration K in a constrained actor theory *cATh* is the set of interaction paths associated with admissible computations of K. The interface of the interaction path *ip* associated with a computation path π is that of the initial configuration of π and its interaction sequence is the subsequence of input/output transition labels of π. Thus it hides internal state and transitions. When the constrained actor theory *cATh* is clear from context we write simply $[\![K]\!]$.

Operations on Actor theories

There are many operations for moving between generalized and specialized actor theories, and for combining theories to make richer theories. These operations allow us to move around in the space of actor theories in semantically meaningful ways and serve as an important part of a toolkit for reasoning about actor system components. Here we describe only what is needed in our example of reasoning about specification diagrams: isomorphism, state restriction, product, and localization.

Isomorphism. Two (constrained) actor theories are *isomorphic* if there are bijections between their states and rules such that corresponding rules have corresponding states and the same triggers and effects, and such that the bijections lift to a bijection between the admissible paths that preserves interaction labels.

Lemma 6 (iso): If $cATh_0$ and $cATh_1$ are isomorphic and $K_0 : cATh_0$ corresponds to $K_1 : cATh_1$, then $[\![K_0 : cATh_0]\!] = [\![K_1 : cATh_1]\!]$.

State restriction. Restricting an actor theory to use a subset of its states allows us to move from a general to a more specific theory. Let *ATh* be an actor theory and let $\mathbf{S}_0 \subseteq \mathbf{S}$ be a subset of the states of *ATh*. The restriction $ATh\lceil\mathbf{S}_0$ of *ATh* to $\mathbf{S}_0$ is obtained by closing $\mathbf{S}_0$ under rules and renaming and then restricting all operations and rules to the resulting set of states. If $cATh = \langle ATh, \mathcal{A}\rangle$ is a constrained actor theory and $\mathbf{S}_0 \subseteq \mathbf{S}$, then the state restriction of $cATh$ given by $cATh\lceil\mathbf{S}_0 = \langle ATh\lceil\mathbf{S}_0, \mathcal{A}\lceil\mathbf{S}_0\rangle$ where $\mathcal{A}\lceil\mathbf{S}_0$ is the restriction of $\mathcal{A}$ to computations of $ATh\lceil\mathbf{S}_0$.

Lemma 7: State restriction preserves interaction semantics: if $\mathbf{S}_0 \subseteq \mathbf{S}$ and K is a configuration of $cATh\lceil\mathbf{S}_0)$, then $[\![K : cATh\lceil\mathbf{S}_0]\!] = [\![K : cATh]\!]$.

Product. The product of two actor theories $cATh_0$, $cATh_1$ (over the same set of actor names and messages) is the actor theory $cATh_0 \times cATh_1$ formed as follows: the states of the product theory are pairs of states $(\sigma_0, \sigma_1) \in \mathbf{S}_0 \times \mathbf{S}_1$ with disjoint internal actors; the reaction rules of the product theory are the union of the rules of the component theories lifted to pairs by acting on the appropriate component; and the admissible paths $\mathcal{A}_{01}$ of the product theory are those paths that project to admissible paths in the component theories. A useful property of product is that it commutes with interaction semantics preserving transformations.

Lemma 8: Let $cATh_j$, $cATh'_j$, and $K_j = \langle\sigma_j, \mu_j\rangle^{\rho_j}_{\chi_j}$ be s.t. $[\![K_j : cATh_j]\!] = [\![K_j : cATh'_j]\!]$ for $j < 2$. Assume $iacts(\sigma_0) \cap iacts(\sigma_1) = \emptyset$ and let $\rho = \rho_0 \cup \rho_1$, $\chi = \chi_0 \cup \chi_1 - \rho$, and $K = \langle(\sigma_0, \sigma_1), \mu_0 \cdot \mu_1\rangle^{\rho}_{\chi}$. Then $[\![K : cATh_0 \times cATh_1]\!] = [\![K : cATh'_0 \times cATh'_1]\!]$.

Localizing messages. When actor systems are combined, many interactions with the environment become internal exchanges of messages that are no longer visible in the interaction semantics. This happens for example when the product of two or more actor theories is formed. We can make these interactions silent in the computation path semantics by applying the *localization* transformation. The localization $Loc(ATh)$ of an actor theory ATh is obtained by adding an internal mail buffer to each state and putting messages sent by internal actors to internal actors in this buffer directly rather than in the undelivered message sets μ. The rules are correspondingly modified to take messages both from the internal buffer and from the undelivered message set. Localization lifts naturally to computation paths and the admissible paths of a localized constrained actor theory are the localizations of admissible paths of the original theory.

Lemma 9 (Localization): Let $cATh$ be a constrained actor theory with configuration K, and K' be a corresponding localized configuration of $Loc(cATh)$; then, $[\![K : cATh]\!] = [\![K' : Loc(cATh)]\!]$.

OPERATIONAL SEMANTICS OF DIAGRAMS

Diagrams are given meaning via an operational semantics, sketched here. The goal is to give a set of interaction paths $[\![\langle D\rangle^{\rho}_{\chi}]\!]$ defining the behavior of top-level diagrams $\langle D\rangle^{\rho}_{\chi}$. This is accomplished by defining a constrained actor theory $cSDTh = \langle SDTh, \mathcal{A}^{\mathrm{sd}}\rangle$. which specifies the computation paths of diagrams. To define the diagram actor theory, $SDTh$ the obligations are to define the states and reaction rules. For the constrained diagram actor theory $cSDTh$ the admissibility predicate, $\mathcal{A}^{\mathrm{sd}}$ must also be defined.

Definition 10 ($SDTh$)**:** The states of $SDTh$ are of the form $\sigma = \langle\!| D,\ iA |\!\rangle$, where D is a diagram syntactically indicating the current state of execution, and iA are the actors defined as internal to D. The acquaintance and internal actors operations are defined by $acq(\langle\!| D,\ iA |\!\rangle) = acq(D) \cup acq(iA)$, and $iacts(\langle\!| D,\ iA |\!\rangle) = iA$. The labelled reaction rules RR^{sd} are a rigorous version of the informal descriptions given in the Syntax section.

See [18, 17] for the complete ruleset RR^{sd}. Here we illustrate the rules via an example computation. Recall the function composer system FC of the Examples Section. We will show one complete pass through the loop. For this purpose we define two additional diagrams that correspond to correspond to positions in the unfolding of the inital state diagram using the environment γ to specify currently bound variables. Let γ contain the bindings $\{x \mapsto v, y \mapsto w, xc \mapsto c, xf \mapsto cf, xg \mapsto cg\}$ and we define

$$
\begin{aligned}
&\mathrm{FC1}(a,af,ag,\gamma) = \{x \mapsto v, xc \mapsto c, xf \mapsto cf:\\
&\quad \mathtt{receive}(xf \lhd \mathtt{reply}(y)); \mathtt{fresh}(xg); \mathtt{send}(ag \lhd \mathtt{compute}(y)@xg);\\
&\quad \mathtt{receive}(xg \lhd \mathtt{reply}(z)); \mathtt{send}(xc \lhd \mathtt{reply}(z)); \mathrm{FC}(a,af,ag)\}\\
&\mathrm{FC2}(a,af,ag,\gamma) = \{x \mapsto v, xc \mapsto c, xf \mapsto cf, y \mapsto w, xg \mapsto cg:\\
&\quad \mathtt{receive}(xg \lhd \mathtt{reply}(z)); \mathtt{send}(xc \lhd \mathtt{reply}(z)); \mathrm{FC}(a,af,ag)\}
\end{aligned}
$$

In each state the internal actors a, A consists of the initially present actor a and the actors created to receive replies A.

An example of reaction steps via the reaction rules is then (recalling $\mathtt{[}D\mathtt{]}^{0\ldots\infty}$ abbreviates $\mathtt{rec}X.((D;X) \oplus \mathtt{skip})$) is:

$$
\begin{aligned}
&\langle\!| \mathrm{FC}(a,af,ag),\ a,A |\!\rangle\\
&\qquad \xrightarrow{\mathtt{rec}(X)} \xrightarrow{\mathtt{choose}(l)} \xrightarrow{\mathtt{receive}(a\lhd\mathtt{compute}(v)@c)} \xrightarrow{\mathtt{seq}} \xrightarrow{\mathtt{fresh(cf)}} \xrightarrow{\mathtt{seq}} \xrightarrow{\mathtt{send}(af\lhd\mathtt{compute}(v)@cf)} \xrightarrow{\mathtt{seq}}\\
&\langle\!| \mathrm{FC1}(a,af,ag,\gamma_1),\ a,cf,A |\!\rangle \quad \text{where} \quad \gamma_1 = \{x \mapsto v, xc \mapsto c, xf \mapsto cf\}\\
&\langle\!| \mathrm{FC1}(a,af,ag,\gamma_1),\ a,cf,A |\!\rangle \xrightarrow{\mathtt{receive}(cf\lhd\mathtt{reply}(w))} \xrightarrow{\mathtt{seq}} \xrightarrow{\mathtt{fresh(cg)}} \xrightarrow{\mathtt{seq}} \xrightarrow{\mathtt{send}(ag\lhd\mathtt{compute}(w)@cg)} \xrightarrow{\mathtt{seq}}\\
&\langle\!| \mathrm{FC2}(a,af,ag,\gamma_2),\ a,cg,cf,A |\!\rangle \quad \text{where} \quad \gamma_2 = \gamma_1\{y \mapsto w, xg \mapsto cg\}\\
&\langle\!| \mathrm{FC2}(a,af,ag,\gamma_2),\ a,cg,A |\!\rangle \xrightarrow{\mathtt{receive}(cg\lhd\mathtt{reply}(u))} \xrightarrow{\mathtt{seq}} \xrightarrow{\mathtt{send}(c\lhd\mathtt{reply}(u))} \xrightarrow{\mathtt{seq}}\\
&\langle\!| \mathrm{FC}(a,af,ag),\ a,cg,A |\!\rangle
\end{aligned}
$$

The `seq` steps in the above moves on to the next diagram expression in the sequence.

Lemma 11: *SDTh* as defined above is an actor theory.

We now define the admissibility predicate to give a constrained actor theory. Recall from the definitions that in the actor theory *SDTh*, configurations K are of the form $\langle\langle\!| D,\ iA |\!\rangle, \mu\rangle^{\rho}_{\chi}$. Thus computation paths $\pi \in \mathcal{P}$ are of the form

$$
\pi = [\langle\langle\!| D_i,\ iA_i |\!\rangle, \mu_i\rangle^{\rho_i}_{\chi_i} \xrightarrow{tl_i} \langle\langle\!| D_{i+1},\ iA_{i+1} |\!\rangle, \mu_{i+1}\rangle^{\rho_{i+i}}_{\chi_{i+1}} \bigm| i \in \mathbf{Nat}].
$$

Definition 12 (*cSDTh*): *cSDTh* is defined as the constrained actor theory $\langle SDTh, \mathcal{A}^{\mathrm{sd}}\rangle$ where $\mathcal{A}^{\mathrm{sd}}$, the admissibility predicate on paths, is defined the restriction to paths which (1) do not get stuck at any point (for instance by failing a `constrain` or waiting for a message that never arrives) and (2) eventually process all messages in μ.

Given the definitions of the reaction rules and admissibility predicate, it is now possible to "turn the crank" using the actor theory framework of the Actor Theory Framework Section and produce interaction semantics for diagrams:

Definition 13: The interaction path semantics of a top-level diagram, $[\![\langle D\rangle^{\rho}_{\chi}]\!]$, is then defined as $[\![\langle\langle\!| \{D\},\ dInActs(D,\rho,\chi) |\!\rangle, \emptyset\rangle^{\rho}_{\chi} : cSDTh]\!]$, where the initial internal actors $dInActs(D,\rho,\chi) = (\rho \cup acq(D) - \chi)$.

Diagrams that contain `assert` may be checked to determine whether the assertions are valid in all paths.

Definition 14: A diagram is (truth-) valid, $\models \langle D\rangle^{\rho}_{\chi}$, iff there are no $\pi \in \mathcal{A}^{\mathrm{sd}}$ with initial configuration $\langle \langle\!| \{D\},\ dInActs(D,\rho,\chi) |\!\rangle, \emptyset\rangle^{\rho}_{\chi}$ that contain an $\mathtt{assert}(\mathit{false})$-labelled transition.

Canonical and Macro-Step Forms. We next develop simpler *canonical forms* for diagram computations to make it easier to reason about them.As the operational semantics is currently structured, atomic computation steps such as `seq`, `choose`, and `rec` are very small units of work. When reasoning about parallel threads, at first glance all possible interleavings of such steps need to be considered. To reduce the number of interleavings, we group atomic steps into *big steps*, a path segment consisting of atomic steps which can without loss of generality be performed in immediate sequence. The *canonical form computations* are those that only perform maximal big steps. Steps beyond the trivial ones listed may also be grouped in big steps. The general criterion is that a big step must be oblivious of any steps that could be interleaved in parallel with its execution. We define $cSDTh_{\mathrm{can}}$ as *cSDTh* with the canonical computations compressed together into macro steps. As an example of a macro step transition in $cSDTh_{\mathrm{can}}$, we revisit the FC example execution covered previously, in macro-step form:

$$\langle\!| \mathrm{FC}(a,af,ag),\ a,A |\!\rangle \xrightarrow[cf,af\triangleleft\mathtt{compute}(v)@cf]{a\triangleleft\mathtt{compute}(v)@c} \langle\!| \mathrm{FC1}(a,af,ag,\gamma_1),\ a,cf,A |\!\rangle$$

$$\text{where}\quad \gamma_1 = \{x \mapsto v, xc \mapsto c, xf \mapsto cf\}$$

$$\langle\!| \mathrm{FC1}(a,af,ag,\gamma_1),\ a,cf,A |\!\rangle \xrightarrow[cg,ag\triangleleft\mathtt{compute}(w)@cg]{cf\triangleleft\mathtt{reply}(w)} \langle\!| \mathrm{FC2}(a,af,ag,\gamma_2),\ a,cg,cf,A |\!\rangle$$

$$\text{where}\quad \gamma_2 = \gamma_1\{y \mapsto w, xg \mapsto cg\}$$

$$\langle\!| \mathrm{FC2}(a,af,ag,\gamma_2),\ a,cg,A |\!\rangle \xrightarrow[c\triangleleft\mathtt{reply}(u)]{cg\triangleleft\mathtt{reply}(u)} \langle\!| \mathrm{FC}(a,af,ag),\ a,cg,A |\!\rangle$$

Notice how multiple transitions are replaced by single macro-steps here. Each macro step can have arbitrary internal steps, and at most one receive followed by any number of sends.

When reasoning about specification diagrams it is most convenient to specialize an actor theory for the particular diagram under study. So while up to now we have had a *single* actor theory *cSDTh* for all specification diagrams, we now define *different* actor theories specialized only to a particular diagram execution. For this purpose we define the operation $\Delta(\langle\!| D,\ iA |\!\rangle)$ that specializes $cSDTh_{\mathrm{can}}$ to just the states reachable from initial state $\langle\!| D,\ iA |\!\rangle$.

Definition 15 (specializing): $\Delta(\langle\!| D,\ iA |\!\rangle) = (cSDTh_{\mathrm{can}})\lceil \langle\!| D,\ iA |\!\rangle$.

Lemma 16: $\langle \langle\!| D,\ iA |\!\rangle, \emptyset\rangle^{\rho}_{\chi} : cSDTh \ \models\!\mid\ \langle \langle\!| D,\ iA |\!\rangle, \emptyset\rangle^{\rho}_{\chi} : \Delta(\langle\!| D,\ iA |\!\rangle)$.

In general, the parallel diagram construction $D_0 \mid D_1$ is not compositional with respect to the interaction semantics, because both D_0 and D_1 may specify `receive` actions for the same actor. In the case that two diagrams `receive` on disjoint sets of actor names then the parallel composition of the diagrams corresponds to the product operation on the associated actor theory. We write $D_0 \bowtie D_1$ to indicate this disjointness property.

Lemma 17 (parallel-product): If D_0, D_1 are specification diagrams such that $D_0 \bowtie D_1$ and iA_0, iA_1 are disjoint and contain the receiving actors for D_0, D_1 respectively, then

$$\langle(\!|D_0 \mid D_1,\ iA_0 \cup iA_1|\!),\mu\rangle^{\rho}_{\chi} : cSDTh \models\!\!\!\!\!\!\,= \langle((\!|D_0,\ iA_0|\!),(\!|D_1,\ iA_1|\!)),\mu\rangle^{\rho}_{\chi} : cSDTh \times cSDTh$$

for μ, ρ, χ such that the configurations are well formed.

PROVING SPECIFICATIONS CORRESPOND

We now show how the techniques developed above may be used to establish properties of diagrams. An important form of reasoning about diagrams is to show that a simple, high-level diagram is equivalent to (strongly satisfied by) a diagram that is the composition of component diagrams. As a simple example we consider the function composer presented in the Examples Section. We show how the purely local computation of $g \circ f$ is equivalent to the distributed implementation. This is Theorem 3 of the Examples Section. To prove the Theorem we must show that $[\![\langle \mathrm{C}(a,f,g,af,ag)\rangle^{a}_{\emptyset}]\!] = [\![\langle \mathrm{F}(a,g\circ f)\rangle^{a}_{\emptyset}]\!]$.

First we calculate the specializations of the F and FC diagrams to obtain very simple actor theory descriptions of the semantics. We then apply lemma 17 that allows us to represent the parallel composition in $\mathrm{C}(a,f,g,af,ag)$ by products and lemma 8 that allows us to specialize the individual actor theory descriptions before applying the product. Finally we apply the localization transformation to the product actor theory. This yields an actor theory isomorphic to the specialization for $\mathrm{F}(a,g\circ f)$ which establishes the theorem.

The specialization $\Delta((\!|\mathrm{F}(a,f),\ a|\!))$ has two states, $(\!|\mathrm{F}(a,f),\ a|\!)$ and $(\!|\mathtt{skip},\ a|\!)$, and two rules:

$$(\!|\mathrm{F}(af,f),\ a|\!) \longrightarrow (\!|\mathtt{skip},\ a|\!)$$
$$(\!|\mathrm{F}(af,f),\ a|\!) \xrightarrow[c\triangleleft\mathtt{reply}(f(v))]{a\triangleleft\mathtt{compute}(v)@c} (\!|\mathrm{F}(af,f),\ a|\!)$$

Note that the construction of this actor theory is uniform in the parameters af, f.

The specialization for the function composer $\Delta((\!|\mathrm{FC}(a,af,ag),\ a,af,ag|\!))$ has four families of states and four rules. The states are: the initial state $(\!|\mathrm{FC}(a,af,ag),\ a,A|\!)$; the final state $(\!|\mathtt{skip},\ a,A|\!)$; and two intermediate states $(\!|\mathrm{FC1}(a,af,ag,\gamma),\ a,A|\!)$ and $(\!|\mathrm{FC2}(a,af,ag,\gamma),\ a,A|\!)$. In each state the internal actors a,A consists of the initially present actor a and the actors A created to receive replies. The intermediate diagrams FC1 and FC2 were given as part of the computation example of the previous section. The rules of $\Delta((\!|\mathrm{FC}(a,af,ag),\ a|\!))$ are $(\!|\mathrm{FC}(a,af,ag),\ a,A|\!) \rightarrow (\!|\mathtt{skip},\ a,A|\!)$ plus the three FC transitions used to illustrate macro steps in the previous section.

Now consider the composition $\mathrm{C}(a,f,g,af,ag)$ defined above. By lemmas 17 and 8, we have

$$\begin{array}{l}\langle \mathrm{C}(a,f,g,af,ag)\rangle^{a}_{\emptyset} : cSDTh \\ \quad \models\!\!\!\!\!\!\,= \\ (\langle(\!|\mathrm{FC}(a,af,ag),\ a|\!),(\!|\mathrm{F}(af,f),\ af|\!),(\!|\mathrm{F}(ag,g),\ ag|\!)\rangle^{a}_{\emptyset} : \\ \qquad\qquad (\Delta((\!|\mathrm{FC}(a,af,ag),\ a|\!)) \times \Delta((\!|\mathrm{F}(af,f),\ af|\!)) \times \Delta((\!|\mathrm{F}(ag,g),\ ag|\!)))\end{array}$$

We localize the product theory giving a theory with states of the form

$$\langle\!\langle \mathrm{CC}_{\mathtt{skip}},\ a, af, ag, A \rangle\!\rangle = (\langle\!\langle \mathtt{skip},\ a, A\rangle\!\rangle, \langle\!\langle \mathtt{skip},\ af\rangle\!\rangle, \langle\!\langle \mathtt{skip},\ ag\rangle\!\rangle, \emptyset)$$

and

$$\begin{aligned}&\langle\!\langle \mathrm{CC}{*}(a, af, ag, f, g, \gamma, \mu),\ a, af, ag, A\rangle\!\rangle = \\ &\quad (\langle\!\langle \mathrm{FC}{*}(a, af, ag, \gamma),\ a, A\rangle\!\rangle, \langle\!\langle \mathrm{F}(af, f),\ af\rangle\!\rangle, \langle\!\langle \mathrm{F}(ag, g),\ ag\rangle\!\rangle, \mu)\end{aligned}$$

where $*$ is the empty string, 1, or 2 and in the empty string case γ is not relevant. Also μ contains messages to actors in $\{af, ag, A\}$, but not to a. Taking the starting state to be $\langle\!\langle \mathrm{CC}(a, af, ag, f, g, \emptyset, \emptyset),\ a, af, ag\rangle\!\rangle$ the restricted set of states and rules is as follows.

$$\begin{aligned}
&\langle\!\langle \mathrm{CC}(a, af, ag, f, g, _, \emptyset),\ a, af, ag, A\rangle\!\rangle \longrightarrow \langle\!\langle \mathrm{CC}_{\mathtt{skip}},\ a, af, ag, A\rangle\!\rangle \\
&\langle\!\langle \mathrm{CC}(a, af, ag, f, g, \emptyset, \emptyset),\ a, af, ag, A\rangle\!\rangle \xrightarrow[cf]{a \triangleleft \mathtt{compute}(v)@c} \\
&\langle\!\langle \mathrm{CC1}(a, af, ag, f, g, \gamma_1, af \triangleleft \mathtt{compute}(v)@cf),\ a, af, ag, cf, A\rangle\!\rangle \longrightarrow \\
&\langle\!\langle \mathrm{CC1}(a, af, ag, f, g, \gamma_1, cf \triangleleft \mathtt{reply}(f(v))),\ a, af, ag, cf, A\rangle\!\rangle \xrightarrow[cg]{} \\
&\langle\!\langle \mathrm{CC2}(a, af, ag, f, g, \gamma_2, ag \triangleleft \mathtt{compute}(f(v))@cg),\ a, af, ag, cf, cg, A\rangle\!\rangle \longrightarrow \\
&\langle\!\langle \mathrm{CC2}(a, af, ag, f, g, \gamma_2, cg \triangleleft \mathtt{reply}(g(f(v)))),\ a, af, ag, cf, cg, A\rangle\!\rangle \xrightarrow[c \triangleleft \mathtt{reply}(g(f(v)))]{} \\
&\langle\!\langle \mathrm{CC}(a, af, ag, f, g, _, \emptyset),\ a, af, ag, cf, cg, A\rangle\!\rangle
\end{aligned}$$

We collapse the receive, silent, and send steps to a single big-step using a variant of the big-step transform and the result is an actor theory that is then isomorphic to $\Delta(\langle\!\langle \mathrm{F}(a, g \circ f),\ a\rangle\!\rangle)$, completing the proof. □

RELATED WORK

A wide variety of notations for concurrent/distributed system specification have been proposed. Specification diagrams share features with many different schools but are still quite separate from existing schools. We very briefly review some of the related approaches here.

Specification diagrams are most closely related to other forms of message-passing diagram, diagrams with vertical lines for processes/threads, and horizontal lines for messages. Message passing diagrams have a long history in software specification and are now most widely known as UML Sequence Diagrams [16]. The Examples Section gave a detailed contrast between specification diagrams and sequence diagrams. In the actor model, event diagrams [8, 10, 3] graphically model scenarios of actor computation by message-passing edges between actors, and were another source of inspiration for this work.

Specification diagrams also share commonalities with other approaches to precise specification. Process algebra notation may be used to formally specify the communication actions of concurrent systems. Parallel composition and choice is of a similar sort in specification diagrams and process algebra. Message send and receive is partly analogous to the related concepts in the asynchronous π-calculus [11]. A number of full specification languages based on process algebra have been developed; examples include LOTOS [2], which is based on CSP; it is now an an ISO standard.

Temporal logic formulae have been extensively used as a means for logical specification of concurrent and distributed systems [12]. Recently temporal logics for distributed object based systems have been developed [7, 5]. While such logics express an extremely broad collection of properties, a significant disadvantage is the need for large, complex formulae to specify nontrivial systems. Specification diagrams themselves can serve the purpose of a logic by directly expressing safety and liveness properties, as was illustrated by the examples. The use of embedded non-computational assertions is very similar to forms found in Dijkstra-style weakest precondition logics for non-concurrent programs [14].

Finite automata are useful for specifying systems which have a strong state-based behavior. The Statecharts automata formalism [9] has become particularly popular in industry. The primary weakness of finite automata is that a complex software system may not have a meaningful global state.

References

[1] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[3] W. D. Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT, 1981. MIT Artificial Intelligence Laboratory AI-TR-633.

[4] Rational Software Corporation. *UML Notation Guide, version 1.1*. September 1997. Obtained From `http://www.rational.com`.

[5] G. Denker. DTL$^+$: A Distributed Temporal Logic Supporting Several Communication Principles. Technical Report , SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA 94025, 1998. *To appear*.

[6] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*, volume 14 of *Texts and Monographs in Computer Science*. Springer-Verlag, 1990.

[7] C. H. C Duarte. A proof-theoretic approach to the design of object-based mobility. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Volume 2*, pages 37–53. Chapman & Hall, 1997.

[8] I. Greif. Semantics of communicating parallel processes. Technical Report 154, MIT, Project MAC, 1975.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[11] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *ECOOP*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1992.

[13] I. A. Mason and C. L. Talcott. Actor languages their syntax, semantics, translation, and equivalence, 1999. to appear.

[14] Greg Nelson. A generalization of dijkstra's calculus. *TOPLAS*, 11:517–561, 1987.

[15] Jim Rumbaugh and Grady Booch. *Unified Method for Object Oriented Development, version 0.8*. 1996. Obtained From `http://www.rational.com`.

[16] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[17] S. Smith. On specification diagrams for actor systems. In C. Talcott A. Gordon, A .Pitts, editor, *Proceedings of the Second Workshop on Higher-Order Techniques in Semantics*, Electronic Notes in Theoretical Computer Science. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume10.html`.

[18] S. Smith and C. Talcott. Specification diagrams for actor systems. See `http://www.cs.jhu.edu/~scott/specdiag`.

[19] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.

[20] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 675–788. North-Holland, Amsterdam, 1990.